

MACHINE ARCHITECTURE

Objectives Of This Module

This module explains the parts of the computer and how they interact. You also will be introduced to assembly language and the various numbering systems used by the computer.

Overview

1. Assembly Language and Machine Code.
This section offers an introduction to assembly language and machine code through a text output routine.
2. Number Systems and Conversion.
Here you will compare the three numbering systems (decimal, hexadecimal, and binary) used by the computer. Exercises in numeric conversion from one base system to another will prepare you for loading and accessing memory.
3. Machine Memory.
This section will answer such questions as: "What is memory? How is it organized? Which parts can I use?"
4. Central Processing Unit.
The 6502 contains six components that actively process information while executing your program. Each component's role in the computing process will be discussed.
5. Additional Chips.
As you probably already know, the Atari features three additional chips to enhance the computer's graphics and sound capabilities. Each will be explained briefly.

Prerequisite Concepts

This is an introductory module intended for the advanced BASIC programmer who wishes to become familiar with microcomputer architecture. An understanding of the BASIC PEEK and POKE commands is necessary.

Materials Needed

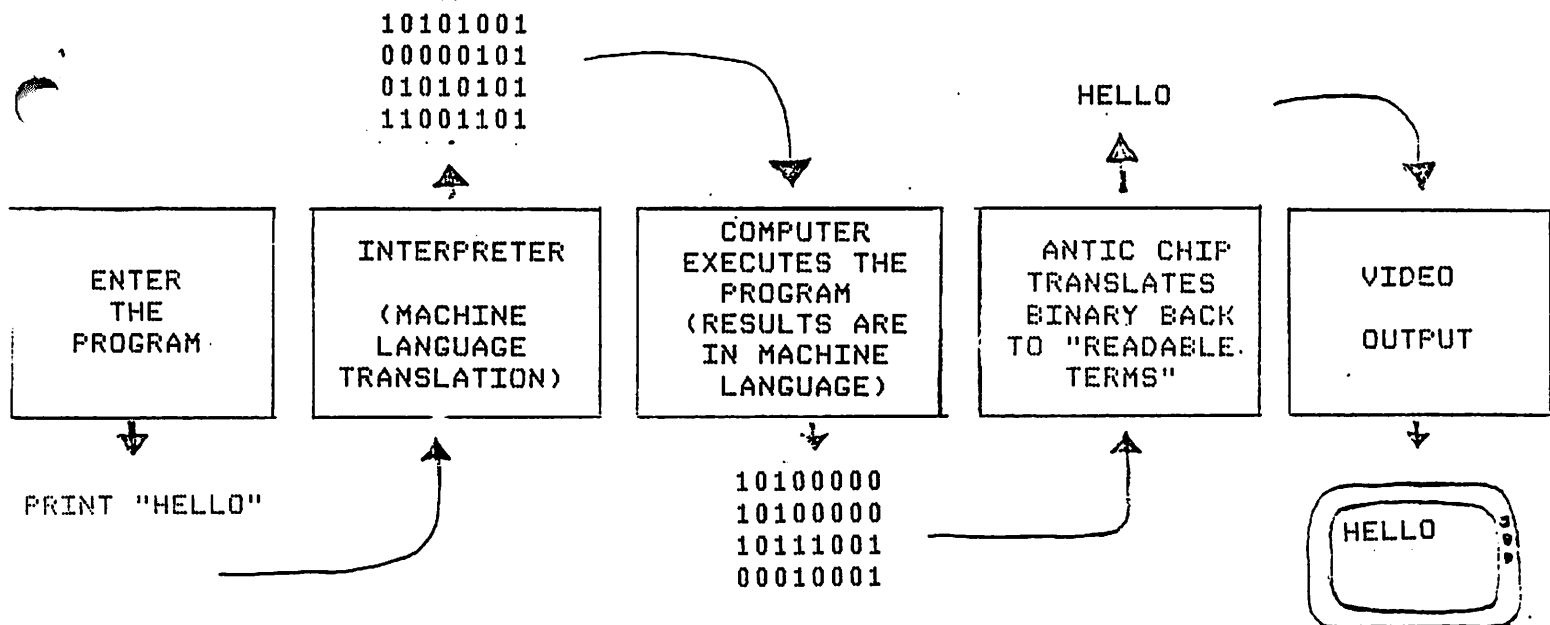
1. A BASIC cartridge.
2. An Advanced Topics Diskette.
3. A Master Memory Map by Santa Cruz Educational Software.

Assembly Language and Machine Code

In this section you will use an assembly language routine to print text on the screen. You will "POKE" the assembly language routine into memory from a BASIC program and run it by using the USR function.

The results of a PRINT statement appear so quickly on the screen that little work seems to be required on the part of the computer. However, in reality, the computer goes through numerous steps merely to print "HELLO" on the screen. Below is a diagram of the steps the computer takes to execute a PRINT statement.

Diagram 1



The second box in the sequence represents the BASIC interpreter. The interpreter is a program that resides in memory and converts your BASIC program to ones and zeros when you RUN your program. Ones and zeros are all your computer really understands. The ones and zeros are grouped together into segments of eight. Each segment of eight ones and zeros means something specific to the computer. For example, when

the computer prints letters on the screen, 00101000 is the code for an "H", and 00100101 is the code for an "E". Each 1 and 0 is called a "bit," and each eight bit series is called a "byte."

1 = A Bit
10101010 = A Byte (8 Bits)

Thus, it is the job of the machine's interpreter to translate your BASIC program into bytes of information that the computer can understand. Programs in all languages must eventually be translated to machine code in order for the processor to execute them.

Now, for purposes of comparison, take a look at the three programs listed in Diagram 2 on the following page. Don't worry if you do not understand the programs. The purpose of Diagram 2 is to show you the differences among BASIC, Assembly Language, and machine language programs. The three programs each print "HELLO" on the screen.

Diagram 2

BASIC PROGRAM

```
10 PRINT "HELLO"
```

ASSEMBLY LANGUAGE PROGRAM

```

10 ;PRINT A MESSAGE TO THE SCREEN BY
20 ;PLACING THE CODED NUMBERS FOR
30 ;EACH CHARACTER DIRECTLY IN SCREEN
40 ;MEMORY      FILE : SCREEN OUTPUT
50 ;*****
60 ;
100      *=$600
A905     110      LDA #5          ;MESSAGE LENGTH
85CD     120      STA $CD        ;STORE COUNTER
130 ;
140 ;$CD IS A FREE BYTE ON THE ZERO PAGE.
150 ;THE MESSAGE LENGTH IS BEING STORED THERE.
160 ;
A000     170      LDY #00        ;COUNTS EACH LETTER
B91106   180 LETTER LDA $611,Y    ;GET THE NEXT LETTER
9158     190      STA ($58),Y    ;PUT LETTER ON SCREEN
C8       200      INY           ;INCREMENT LETTER COUNTER
C4CD     210      CPY $CD        ;THE END OF THE MESSAGE?
D0F6     220      BNE LETTER     ;NO? GET ANOTHER LETTER
60       230      RTS           ;RETURN
28       240      .BYTE 40,37,44,44,47
25 2C 2C 2F

```

MACHINE LANGUAGE PROGRAM (BINARY CODE)

10101001	11001000
00000101	11000100
10000101	11001101
11001101	11010000
10100000	11110110
00000000	01100000
10111001	00101000
00010001	00100101
00000110	00101100
10010001	00101100
01011000	00101111

The three programs perform the same function: they print "HELLO" on the screen. The program at the top of the page is obviously in BASIC. The program at the bottom of the page is machine language. The machine language program lists each specific step the computer must take to print "HELLO". Assembly language was designed because programming ones and zeros is very tedious and leaves lots of room for errors by the programmer. Assembly language enables the programmer to have closer control of the machine than BASIC, because the instructions are more specific and work in more understandable terms than machine language.

Assembly language is made up of three letter instructions that are abbreviations for the command. For example, the second to the last line of the assembly listing in Diagram 2 contains an "RTS" instruction, which tells the computer to "ReTurn from the Subroutine". In this case the return is to a BASIC program. An RTS is comparable to a "RETURN" in BASIC. The instruction just above the RTS, "BNE", stands for "Branch Not Equal to zero," which is similar to an "If ... THEN" statement in BASIC.

Just as a BASIC program must be interpreted, the assembly language program also must be converted to machine language. Look closely at the assembly language routine in Diagram 2. Just to the left of the assembly language program are peculiar combinations of letters and numbers. Notice for example, Line 110 contains A905 to the left of the LDA #5 instruction.

Hexadecimal Machine Code	Line #	Assembly Instruction	Remark
<u>A905</u>	0110	LDA #5	;MESSAGE LENGTH

↑

This seemingly unintelligible notation is the machine language version of the assembly language program in hexadecimal (base 16). Instead of being displayed in ones and zeros (base two), now the machine code is shown in base 16. (This will be explained in more depth in the Number Systems and Conversion section.) A machine language program then can be represented in binary or hexadecimal numbers. The values are the same, but the notation varies, just as "twenty-five" and 25 are different ways of recording the same value. Thus, A905 in the assembly language

version is the same as the 10101001 and 00000101 in the first two lines of the binary code listing in Diagram 2. (Two hexadecimal digits are equal to one binary byte.)

Hexadecimal	Binary	Assembly Instruction
A9 =	10101001	(LDA)
05 =	00000101	(5)

Machine language is the specific set of steps the computer must take to execute the program. It can be represented in binary numbers (base 2) or hexadecimal numbers (base 16).

It is also possible to POKE the decimal values of the machine language version into memory from BASIC. Listed below are the binary, hexadecimal, and decimal equivalents for the machine language routine. This program still prints "HELLO" on the GRAPHICS 0 screen. The first few values in this version, however, are slightly different from those in Diagram #2 to account for running the program from BASIC.

Diagram 3

Machine Code

<u>Binary</u>	<u>Hexadecimal</u>	<u>Decimal</u>
01101000	68	104
01101000	68	104
01101000	68	104
11000101	85	133
11001101	CD	205
10100000	A0	160
00000000	00	0
10111001	B9	185
00010100	14	20
00000110	06	6
10010001	91	145
01011000	58	88
11001000	C8	200
11000100	C4	196
11001101	CD	205
11010000	D0	208
11110110	F6	246
01100000	60	96
00101000	28	40
00100101	25	37
00101100	2C	44
00101100	2C	44
00101111	2F	47

Since assembly language is frequently used to enhance or speed up a BASIC program, we will run the machine language routine from a short BASIC program. Turn to Machine Architecture Worksheet #1.

Machine Architecture Worksheet #1

Your computer should have a BASIC cartridge in it.

1. Load the program on your Advanced Topics diskette entitled "MESSAGE".

Type: LOAD "D:MESSAGE"

When you list the program, the code in Diagram 4 (on the following page), should appear on the screen.

The function of this program is to read the machine code data from lines 440-460, which contain the program that prints "HELLO" on the screen. Since BASIC uses decimal numbers, the machine language version of the assembly language program must be listed in base 10. (If you look back at Diagram 3, you will find a list of the binary, hexadecimal, and decimal equivalents for the machine language version of this program.) The BASIC program reads the machine code data one number at a time and stores it in memory. The USR function turns the computer's attention to the machine language program in memory to be executed. Take a moment to read the comments accompanying the program.

Diagram 4

```

10 REM *                PRINT MESSAGE
15 REM *
20 REM *      THE DECIMAL VALUES FOR AN ASSEMBLY
25 REM *      LANGUAGE SUBROUTINE ARE POKED
30 REM *      INTO MEMORY.  THE ROUTINE PRINTS
35 REM *      HELLO ON THE GRAPHICS 0 SCREEN.
40 REM *      THE INTERNAL CHARACTER SET IS USED.
45 REM *      TO CHANGE THE MESSAGE,  CHANGE
50 REM *      THE DATA ON LINE 460.
70 REM *****
80 REM
90 REM
100 COUNTER = 0:REM INITIALIZE COUNTER FOR MESSAGE LENGTH
110 PROGRAMLEN = 17 :REM PROGRAM LENGTH IS 18 BYTES (0-17)
120 REM *
130 REM *  LINES 160-190 READ THE DATA FOR THE ASSEMBLY
140 REM *  ROUTINE ON LINES 440-450 AND PUT IT IN MEMORY
150 REM *
160 FOR INSTRUCTION = 0 TO PROGRAMLEN
170 READ CODE
180 POKE 1536+INSTRUCTION, CODE
190 NEXT INSTRUCTION
200 REM *
210 REM *  NOW READ THE MESSAGE DATA ON LINE 460
220 REM *  WHEN OUT OF MESSAGE DATA GOTO 320 VIA TRAP
230 REM *
240 READ MESSAGE
250 TRAP 320
260 COUNTER = COUNTER + 1
270 REM *
280 REM *  PUT THE MESSAGE IN MEMORY FOLLOWING ASSEMBLY
290 REM *  ROUTINE
300 POKE 1555+COUNTER, MESSAGE
310 GOTO 240
320 GRAPHICS 0 : PRINT "      ":REM ACCOUNT FOR BASIC BUG
330 REM *
340 REM *  CALL BELOW EXECUTES THE ASSEMBLY ROUTINE
350 REM *  STARTING AT 1536.  THE LENGTH OF THE
360 REM *  MESSAGE IS PASSED TO THE ASSEMBLY
370 REM *  ROUTINE IN COUNTER
380 REM *
390 CALL = USR (1536,COUNTER)
400 REM *
410 REM *  LINE 440 PASSES THE LENGTH
420 REM *  OF THE MESSAGE TO THE ASSEMBLY ROUTINE
430 REM *
440 DATA 104,104,104
450 DATA 133,205,160,0,185,20,6,145,88,200,196,205,208,246,96
460 DATA 40,37,44,44,47
470 REM *  IF NECESSARY YOU CAN EXTEND YOUR MESSAGE ONTO AN
ADDITIONAL DATA LINE

```

2. RUN the MESSAGE program.

Did HELLO appear in the upper left hand corner of your screen?

3. The DATA on line 460 contains the values for the letters that will be printed on the screen. The machine language routine places the values for the letters directly into memory locations which are reserved for the video screen. By storing the letter in screen memory, the letter is put on the screen. Take a few minutes now to construct your own message. First plan your message below. You needn't restrict your message length to the lines provided below. Your message can be up to 60 characters in length.

To POKE values directly into screen memory, as this program does, you must use the decimal values for the internal character set. You will find a chart of the internal character set and its values on Chart #1 at the back of this module. Record the internal character set value below each of the letters in your message. These numbers represent the data for your message. Replace the decimal numbers on line 460, which represent the word HELLO, with the values for your message.

After you have double checked your typing, run the program. Your message should appear in the upper left hand corner of your screen. Experiment with different messages.

At this point you are probably wondering, "Why bother with all these confusing numbers to accomplish something so easily done in BASIC?" Ordinarily, you would not use an assembly language routine just to print letters on the screen. The BASIC PRINT command is much better suited to that purpose. However, when speed is a factor, assembly language is much more appropriate. Turn to Machine Architecture Worksheet #2 for a comparison of the execution speed of an assembly language program with a BASIC program.

Machine Architecture Worksheet #2

1. In the space below write a simple BASIC program that inputs a letter from the keyboard and fills the entire graphics 0 screen with that letter. Write your program to fill 874 locations on the graphics 0 screen. An example of the program is in the "SCREENFULL" file on your advanced topics diskette.

2. Run your program and record how long it takes to fill the screen. _____seconds

Now let's compare your BASIC program with a similar program done in assembly language. Once again, the machine code for an assembly routine will be poked in from BASIC.

3. Run the BASIC program called "FILLSCREEN" on your advanced topics diskette.

Type: RUN "D:FILLSCREEN"

Press any key and then press another. Can you time how fast the screen fills up with a new character? _____

Summary

Because assembly language specifies what the computer needs to do step by step, the code is more detailed and takes much longer to program. However, in some situations the increased speed of an assembly routine outweighs this disadvantage. This is particularly true when animating graphics.

Machine Code: Step by step instructions for the computer, represented in hexadecimal or binary code (or decimal when POKEd into memory from BASIC).

Hexadecimal Numbers (Base 16): Hexadecimal digits range from 0 through F (the letters A through F are equivalent to the decimal numbers 10 through 15) and are commonly used to represent machine language programs.

Binary Numbers (Base 2): The only digits used in base 2 are 1 and 0. Each digit in a binary number is referred to as a bit.

Byte: Eight bits or binary digits make one byte. (eg., 11010111)

Assembly Language: The programming language closest to machine language. Assembly language consists of three-letter abbreviated instructions or mnemonics.

USR: A BASIC function that enables you to run an assembly language routine in memory from a BASIC program.

NUMBER SYSTEMS AND CONVERSION

In this section you will learn how to recognize and represent hexadecimal and binary numbers. It is necessary to understand these numbering systems because hexadecimal numbers are used to access information in memory and the computer does all of its calculations in binary.

All numbering systems follow a similar scheme. The value of a number is based on the sequence of the digits in the number. So, to understand base 2 and base 16, a quick review of what you already know about base 10 may be useful.

In base 10 we know that the digits 0 through 9 are used. The numbers 0-9 are the ten different symbols that have been selected to represent the ten different digits in base 10. In order to represent different numbers, we assign values to each location a digit occupies. For example, consider the three digit number 768. The 8 is in what we call the ones column. So we have 8 ones. To the left of the ones column is the tens place. There are 6 tens in 768. And of course the 7 is in the hundreds place. Since we are in base 10, each column to the left is ten times the value of the previous column.

Place Values	100's	10's	1's
768	7	6	8
	\wedge	\wedge	\wedge
	7 * 100	6 * 10	8 * 1
	\vee	\vee	\vee
	700	60	8
	+	+	= 768

Hexadecimal Numbers (Base 16)

In base 16, in order to have 16 different symbols to represent the 16 different digits, the numbers 0 through 9 and the letters A to F are used. Thus, in base 16 the digits range from zero through F. The letters A through F represent the decimal numbers 10-15. Below is a list of the hexadecimal digits and their base 10 equivalents.

Hexadecimal	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
Decimal	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

The decimal number sixteen is \$10 in hexadecimal. The dollar sign preceeding the number indicates that the value is in base 16. In base 16 the place values are sixteen times the preceeding place value.

Hexadecimal Place Values	16^3	16^2	16^1	16^0
Decimal Equivalents	4096	256	16	1

The rightmost column is the ones column. The next column to the left holds the number of sixteens in the total value. The third column to the left holds the number of 256's (or 16^2 's) and so on. Consider the example below:

Place Values	4096's	256's	16's	1's	
\$600	$\begin{array}{c} 0 \\ \swarrow \quad \searrow \\ 0 \times 4096 \\ \swarrow \quad \searrow \\ 0 \end{array}$	$\begin{array}{c} 6 \\ \swarrow \quad \searrow \\ 6 \times 256 \\ \swarrow \quad \searrow \\ 1536 \end{array}$	$\begin{array}{c} 0 \\ \swarrow \quad \searrow \\ 0 \times 16 \\ \swarrow \quad \searrow \\ 0 \end{array}$	$\begin{array}{c} 0 \\ \swarrow \quad \searrow \\ 0 \times 1 \\ \swarrow \quad \searrow \\ 0 \end{array}$	= 1536

+ + +

Multiplying each digit by its place value and adding up the products gives you the decimal equivalent to a hexadecimal number. The sum 1536 is the decimal memory location we used for the assembly routine. So the program was stored at \$600 in memory. Look over this next example:

Place Values	4096's	256's	16's	1's	
\$9C40	$\begin{array}{c} 9 \\ \swarrow \quad \searrow \\ 9 \times 4096 \\ \swarrow \quad \searrow \\ 36864 \end{array}$	$\begin{array}{c} C \\ \swarrow \quad \searrow \\ 12 \times 256 \\ \swarrow \quad \searrow \\ 3072 \end{array}$	$\begin{array}{c} 4 \\ \swarrow \quad \searrow \\ 4 \times 16 \\ \swarrow \quad \searrow \\ 64 \end{array}$	$\begin{array}{c} 0 \\ \swarrow \quad \searrow \\ 0 \times 1 \\ \swarrow \quad \searrow \\ 0 \end{array}$	= 40000

+ + +

The starting address of memory for the graphics 0 screen is 40000 or \$9C40. Turn to Machine Architecture Worksheet #3 for some practice problems on hexadecimal to decimal conversion.

Machine Architecture Worksheet #3

Hexadecimal to Decimal Conversion

1. Convert \$173F to a decimal number.

Place Values	4096's	256's	16's	1's	
\$173F	$\begin{array}{c} 1 \\ \diagup \quad \diagdown \\ \text{---} * 4096 \end{array}$	$\begin{array}{c} \text{---} \\ \diagup \quad \diagdown \\ 10 * 256 \end{array}$	$\begin{array}{c} 3 \\ \diagup \quad \diagdown \\ \text{---} * 16 \end{array}$	$\begin{array}{c} F \\ \diagup \quad \diagdown \\ F * 1 \end{array}$	
	-----	+	-----	+	-----

					= 6719

2. Convert the following hexadecimal numbers to decimal numbers.

- | | |
|---------------|----------------|
| 1. \$23 ----- | 9. \$00 ----- |
| 2. \$6F ----- | 10. \$77 ----- |
| 3. \$6D ----- | 11. \$B7 ----- |
| 4. \$70 ----- | 12. \$BA ----- |
| 5. \$75 ----- | 13. \$A1 ----- |
| 6. \$74 ----- | 14. \$B2 ----- |
| 7. \$65 ----- | 15. \$A4 ----- |
| 8. \$62 ----- | |

3. The numbers you just converted to decimal numbers were selected because they represent the letters of a message. To find out what the message is, you will use your answers from problem number 2 as DATA in the MESSAGE program. First, load the file called "MESSAGE" on your Advanced Topics diskette.

Type: LOAD "D:MESSAGE"

Now type in the decimal numbers from problem 2 on line 460 of the MESSAGE program. Leave the word DATA at the beginning of the line, but replace the numbers for "HELLO" with your decimal answers from above. Be sure to separate each number with a comma, starting with your answer to #1 and ending with your answer to #15. If your answers are correct, you should get a message in the upper left hand corner of the screen.

4. Can you write a BASIC program to do conversions from hexadecimal to decimal for you?

NOTE: The hexadecimal and binary numbering systems are quite confusing at first. The more experience you have using each of the two systems, the easier they will be to understand.

Decimal to Hexadecimal Conversion

Finding the hexadecimal equivalent to a decimal number is a little more difficult. But as you begin to write assembly routines, you need to identify where your routine will go in memory. In assembly language, locations in memory are identified with hexadecimal numbers. In BASIC you give the memory location in decimal.

Let's begin by looking over an example of converting the decimal number 40000 to a hexadecimal number. You already know the answer from the hexadecimal to decimal conversion section, so you will know if your calculations are on track.

First, divide 40000 by 16.

$$\begin{array}{r} 2500 \\ 16 \overline{) 40000} \\ \underline{32} \\ 80 \\ \underline{80} \\ 00 \\ \underline{00} \\ 00 \\ \underline{00} \\ 00 \\ \underline{00} \\ 00 \end{array} \quad \begin{array}{l} \$ _ _ _ _ \\ \nearrow \end{array}$$

The remainder goes in the right most column of the hexadecimal value. So far, we have ? ? ? 0. Now divide 2500, the answer you got from your previous division, by 16. Complete the division.

$$\begin{array}{r} 15 \\ 16 \overline{) 2500} \\ \underline{16} \\ 90 \end{array} \quad \begin{array}{l} \$ _ _ \underline{4} \underline{0} \end{array}$$

You should have gotten 156, with a remainder of 4. The 4 goes in the second column from the right, the 16's column.

Divide again! This time, 156 is divided by 16. Finish the problem.

$$\begin{array}{r}
 9 \\
 16 \overline{) 156} \\
 \underline{144} \\
 12
 \end{array}
 \qquad
 \$ _ _ \underline{4} \underline{0}$$

The remainder of 12 (a "C" in hexadecimal) goes in the 256's column. Finally, since 9 is not divisible by 16, it goes in the next column to the left, the 4096's column. So $40000 = \$9C40$.

To check your answer simply multiply each digit by its place value and add.

Place Values

	4096's	256's	16's	1's	
\$9C40	$ \begin{array}{c} 9 \\ \swarrow \quad \searrow \\ 9 \times 4096 \\ \swarrow \quad \searrow \\ 36864 \end{array} $	$ \begin{array}{c} C \\ \swarrow \quad \searrow \\ C \times 256 \\ \swarrow \quad \searrow \\ 3072 \end{array} $	$ \begin{array}{c} 4 \\ \swarrow \quad \searrow \\ 4 \times 16 \\ \swarrow \quad \searrow \\ 64 \end{array} $	$ \begin{array}{c} 0 \\ \swarrow \quad \searrow \\ 0 \times 1 \\ \swarrow \quad \searrow \\ 0 \end{array} $	$ + \quad + \quad + \quad + \quad = \quad 40000 $

Calculate the decimal to hexadecimal conversions on Machine Architecture Worksheet #4.

Machine Architecture Worksheet #4

Decimal to Hexadecimal Conversion

1. Complete the following conversion. To check your answer, simply multiply each digit of the hexadecimal number by its place value and add up the products.

$$2598 = \$ _ _ _ _ _$$

$$\begin{array}{r} 16 \overline{) 2598} \\ \underline{16} \\ 99 \\ \underline{96} \\ 38 \\ \underline{32} \\ 6 \end{array} \quad 16 \overline{) \quad \quad \quad} \quad 16 \overline{) \quad \quad \quad}$$

2. Convert the following decimal numbers to hexadecimal.

$$133 = _ _ _ _ _ _$$

$$205 = _ _ _ _ _ _$$

$$160 = _ _ _ _ _ _$$

$$0 = _ _ _ _ _ _$$

$$185 = _ _ _ _ _ _$$

The decimal numbers you just converted to hexadecimal were the first six numbers in the DATA statement on line 450 of the MESSAGE program. To check your answers look back at the machine code for the MESSAGE program in Diagram 3.

Hexadecimal Place Values	16^3	16^2	16^1	16^0
Decimal Equivalents	4096	256	16	1

Binary numbers are the specific information the computer uses to perform its functions. Base 2 requires only two different symbols for digits. The digits used in base two are zeros and ones. Just as in hexadecimal and decimal notation, the total value of a number is based on the placement of the digits. The place value of each adjacent column to the left is increased exponentially by one, as shown below.

Binary Place Values	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
Decimal Equivalents	128	64	32	16	8	4	2	1

To get the decimal equivalent to a binary number, you simply multiply the digit by its place value and sum up the products, as you did when converting a hexadecimal value to decimal. Study the example of binary to decimal conversion below.

10011010

0	*	1	=	0
1	*	2	=	2
0	*	4	=	0
1	*	8	=	8
1	*	16	=	16
0	*	32	=	0
0	*	64	=	0
1	*	128	=	128
				<u>154</u>

Now turn to Machine Architecture Worksheet #5 and complete the conversions.

Machine Architecture Worksheet #5

1. Convert the following binary numbers to decimal. The table of binary place values at the bottom of the page should be useful.

11111111 = _____ The largest number that can be represented by one byte.

00100001 = _____ 00111010 = _____

01100001 = _____ 01111010 = _____

10100001 = _____ 10111010 = _____

11100001 = _____ 11111010 = _____

2. Load the MESSAGE file on your Advanced Projects diskette.

Type: LOAD "D:MESSAGE"

Type in the decimal numbers you got in the above conversions as data on line 460. Remember to put commas between the decimal values. Then RUN the program.

You should see various forms of the letters "A" and "Z" in the upper right hand corner of your screen.

3. Record the letter as it appears on the screen next to the corresponding decimal number you got at the top of this worksheet.

4. Look back at the binary numbers you converted to get the different "A's". Notice that the only difference in the bits (ones and zeros) of the four numbers are the two bits on the far left. This is also true for the binary numbers you used for the various "Z"'s you got. Those two bits are referred to as bit 6 and 7.

The positions of the digits in a binary value are numbered from zero to seven starting on the right.

Place Values	128	64	32	16	8	4	2	1
Bits	7	6	5	4	3	2	1	0

A bit is said to be "set" if there is a one in the corresponding bit location. Otherwise, the bit is "clear," which means it contains a zero.

In the example above you can see that the value the computer uses for a normal capital "A" is 33. Adding 64 to the 33 sets bit six of the bit pattern for a normal "A", because the place value for bit 6 is 2^6 or 64.

```
00100001 = 33 = A
01100001 = 97 = a
  \ / \ /
  64  33
```

This new value indicates to the computer that you want to print a lower case "a". Adding 197 to the base number for an "A" or setting both bit six and bit seven results in the value for an inverse "a".

11100001 = 197 = Inverse "a"

The base value of 33 for an "A" and the corresponding bit pattern for the letter remain the same. Bits 0-6 stand for the letter "A" and remain unchanged.

To learn more about how the computer knows to print the letter "A" based on the number 33, see the Internal Representation of Text and Graphics Module.

5. You need to be able to convert both ways between two numbering systems, regardless of which base systems you are using. Try converting the following decimal numbers to binary.

- | | |
|----------------|-----------------|
| 1. 133 = _____ | 6. 20 = _____ |
| 2. 205 = _____ | 7. 6 = _____ |
| 3. 160 = _____ | 8. 145 = _____ |
| 4. 0 = _____ | 9. 88 = _____ |
| 5. 185 = _____ | 10. 200 = _____ |

The decimal values you just converted to binary are the same numbers you typed in for DATA on line 460 of the MESSAGE program to POKE in the assembly language routine. Your binary answers should correspond to the fourth through the twelfth numbers listed in the binary code in Diagram #3 towards the beginning of this module. Start with the fourth because the first three are all the same (104).

Binary to Hexadecimal Conversion

Four binary digits are equal to one hexadecimal digit. The highest value that can be represented by four binary digits is 15.

$$1111 = 15 = \$F$$

4 Bits = 1 Hexadecimal Digit

Eight binary digits are equal to two hexadecimal digits or one byte.

$$11111111 = \$FF = \text{one byte}$$

Four bits or half a byte is called a "nybble."

$$1111 = \text{one nybble} = \text{one hexadecimal digit} = \$F$$

To convert a binary nybble to a hexadecimal number, just multiply the digit by its place to get the decimal number, and convert the decimal number to hexadecimal as shown below.

$$\begin{array}{rcll} 1101 & & & \\ \downarrow & \rightarrow & 1 \times 1 & = 1 \\ \downarrow & \rightarrow & 0 \times 2 & = 0 \\ \downarrow & \rightarrow & 1 \times 4 & = 4 \\ \downarrow & \rightarrow & 1 \times 8 & = 8 \\ & & 13 & = \$D \end{array}$$

To convert an eight bit byte to hexadecimal, you simply split the byte into two nybbles and treat each nybble as having place values of 0-8. For example:

$$\begin{array}{c} 10010101 \\ \swarrow \quad \searrow \\ 9 \quad 5 \\ \swarrow \quad \searrow \\ \$95 \end{array}$$

Try converting the binary numbers listed below. Check your answers with a friend.

$$10010101 = \underline{\hspace{2cm}}$$

$$11111110 = \underline{\hspace{2cm}}$$

$$00101111 = \underline{\hspace{2cm}}$$

$$00101100 = \underline{\hspace{2cm}}$$

Summary

1111 = 4 Bits
4 Bits = One Nybble

1111 = \$F
4 Bits = One Hexadecimal Digit

11111111 = 8 Bits
8 Bits = One Byte

11111111 = \$FF
One Byte = Two Hexadecimal Digits

BINARY, HEXADECIMAL, AND DECIMAL EQUIVALENTS

<u>Binary</u>	<u>Hexadecimal</u>	<u>Decimal</u>
00000001	\$01	1
00000010	\$02	2
00000011	\$03	3
00000100	\$04	4
00000101	\$05	5
00000110	\$06	6
00000111	\$07	7
00001000	\$08	8
00001001	\$09	9
00001010	\$0A	10
00001011	\$0B	11
00001100	\$0C	12
00001101	\$0D	13
00001110	\$0E	14
00001111	\$0F	15
11110000	\$F0	3840
11111111	\$FF	65535

A BASIC program which converts decimal numbers to hexadecimal and hexadecimal numbers to decimal is listed in Appendix H of the ATARI BASIC Reference Manual.

At the back of the module is a small conversion chart for hexadecimal to decimal conversion, which may also prove to be useful.

Machine Memory

This section covers how the memory is organized, what it consists of, and how you can access it.

Memory in a computer can be thought of as a long, LONG stack of mail boxes. Each mail box can hold only one standard size envelope. In the Atari, each memory location holds one byte of information. When an assembly program is converted to machine code, each byte of the program is put into successive memory locations.

Assembly	Binary	Memory

		10101001
LDA #5	10101001	-----
STA \$CD	00000101	00000101
	01010101	-----
	11001101	01010101

		11001101

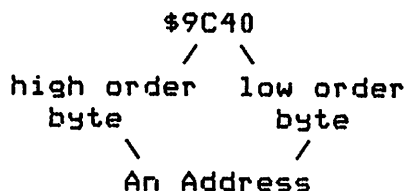
To return to the mail box analogy, each mail box or memory location has an "address." An address is a hexadecimal number that identifies a mail box or memory location. All memory addresses are two bytes long. The first mail box in memory has an address of \$0000. The next box's address is \$0001. In a 64 K computer the address of the last box in memory is \$FFFF.

Addresses	Memory
\$0000	-----
\$0001	-----
\$0002	-----
\$0003	-----
\$0004	-----

\$FFFF	-----

As you may have gathered from the Number Systems and Conversion section, \$FFFF (or its binary equivalent 1111111111111111) is the largest two byte hexadecimal number you can represent. If you were to add 1 to \$FFFF, you would get \$10000, which is a three byte number. All memory addresses in the Atari are two bytes.

Names are given to each of the two bytes that make up a memory address. Take the memory address \$9C40. The two digits on the right, 40, are called the low order byte of the address. The 9C is called the high order byte.

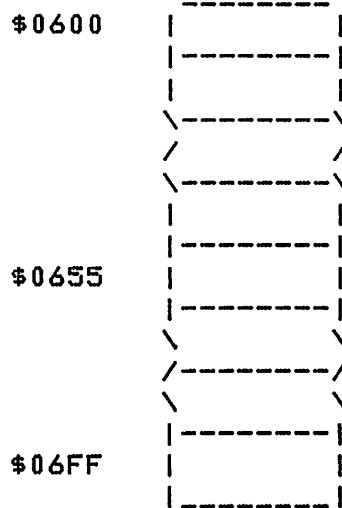


Instead of dealing with lots of individual boxes, convenient systems for dealing with memory in blocks have been devised. One "page" of memory is made up of 256 bytes or memory locations. The addresses on the first page of memory range from \$0000 - \$00FF (\$FF = 255). This is called the zero page. The high order byte of an address on the zero page is always \$00.

\$0000 - \$00FF The high order byte is always \$00
for addresses on the zero page.

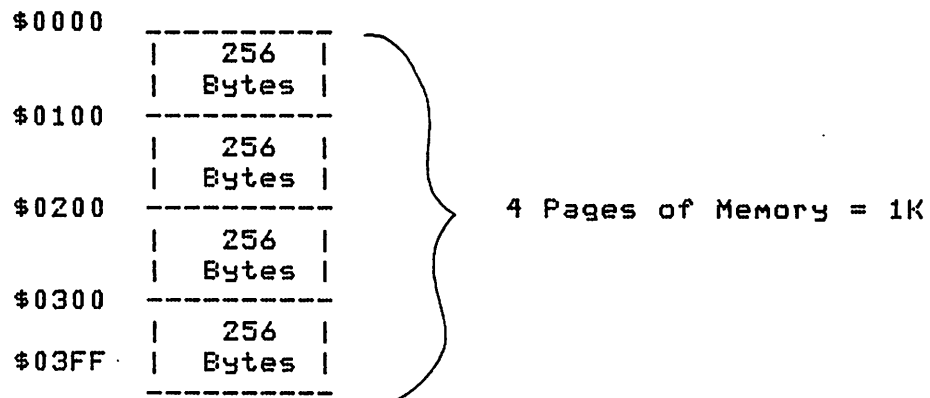
The addresses on page one of memory range from \$0100 - \$01FF. Therefore, the high order byte of all the addresses on page one is \$01. The low order byte of the address indicates which of the 256 memory locations on that page of memory. The address \$0655 is at the 55th location on page six in memory. See the diagram below.

PAGE 6 of Memory



Have you ever wondered why some programs include statements like `Screen = PEEK(89)*256+PEEK(88)?` In this case locations 88 and 89 contain the starting address of screen memory. Since addresses are two bytes long and individual memory locations can only hold one byte, addresses must be stored in consecutive memory locations. Location 89 holds the high order byte of the address. Location 88 contains the low order byte of the address. Since the high order byte of an address is the same as the page number, it is multiplied by the number of bytes on a page.

Chunks of memory are most commonly referred to in terms of the number of "K" of memory you have. The Atari 800 can access 64K of memory when all the memory slots are filled. What exactly is a "K" of memory? Four pages of memory are equal to 1K of memory.



To the programmer, 1K of memory is actually 1,024 bytes of memory locations, not 1,000.

256	1 page of memory = 256 bytes or locations
<u>X 4</u>	4 pages of memory
1024	1K of memory

A 48K computer has 49,152 memory locations.

1024	1K of memory
<u>X 48</u>	Number of K
49,152	Memory locations

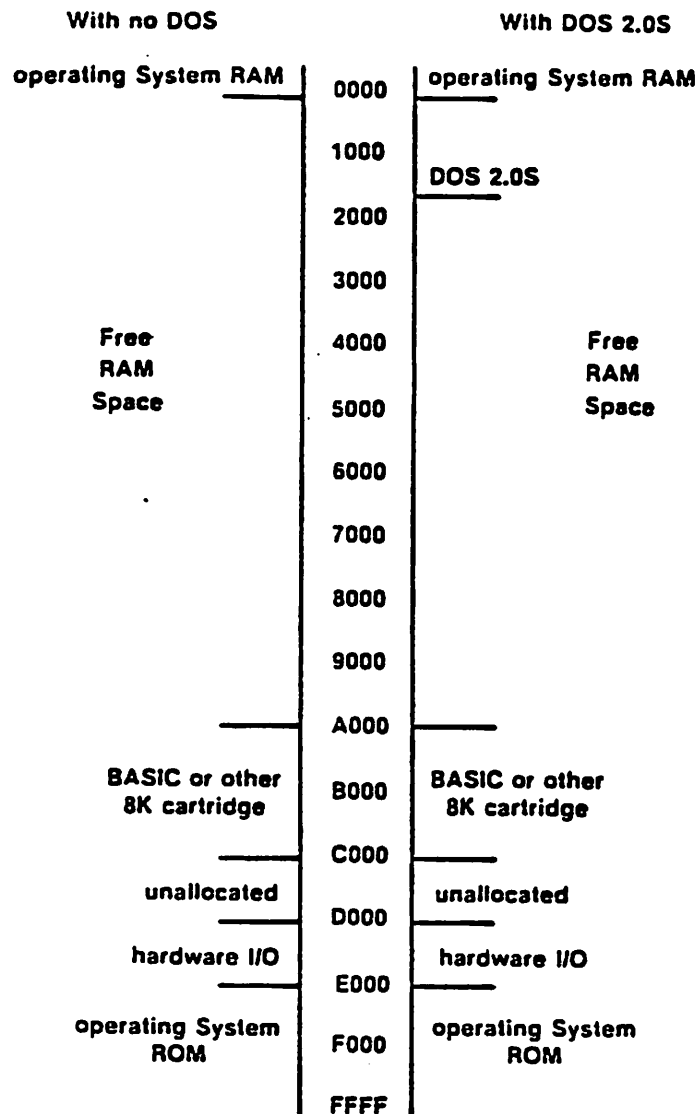
And a 64K machine has 65,536 memory locations.

1024	1K of memory
<u>X 64</u>	Number of K
65536	Memory locations in 64K

If you count the first memory location as zero, the memory locations are numbered 0 through 65,535. The number 65,535 probably looks familiar. If you convert the decimal number 65,535 to hexadecimal, you will find that it equals \$FFFF, the highest possible address in memory.

Not all 65,536 memory locations are available to you, the programmer. The memory which the programmer can change is called "RAM" or Random Access Memory. Random access means that you can store anything you wish in those locations as well as read data from them. Random access memory also is referred to as "read write memory." You can think of Read Write memory as being like a blackboard which you can write on, read from, or erase and start over. Take a look at the memory map in Diagram 6 on the next page. Right away you can see that there is quite a bit of free RAM available to you.

Diagram 6



The areas of memory that are not free for the programmer to use are called "ROM" or Read Only Memory. ROM contains machine language programs that enable you to speak to your computer and get information back. For example, the machine code for the letters and symbols printed on the screen are stored in ROM. The "Operating System," which includes programs that interpret keypresses and provide access

to the various input and output devices, also reside in ROM. DOS, or the Disk Operating System programs, which handle loading, copying, and saving your files on disk, also resides in ROM. The computer will not let you store any of your own programs in ROM, because you would destroy the routines which enable the computer to process your program. Instead, the routines in ROM are read and used continually as you use your computer. There is nothing a programmer can do which will damage or change ROM unintentionally. ROM read only memory can be thought of as a book inside the computer. You can read from it, but you cannot change the print or the information in the book. Turn to Machine Architecture Worksheet #6 to look at the contents of a portion of RAM and ROM.

Machine Architecture Worksheet #6

You will need the Assembler Editor cartridge and the Advanced Topics diskette to complete this worksheet.

Insert the Assembler Editor cartridge in slot A, where you ordinarily put your BASIC cartridge. Turn off your computer and boot your Advanced Topics diskette. The word "EDIT" should be in the upper left hand corner of your screen. If not, reboot the system. Now you are ready to begin.

Although your computer has 65,536 memory locations, they are not all available to you. The memory available to the programmer is called "RAM" or Random Access Memory. Random access memory is storage space in memory for your programs.

1. First, let's look at the contents of some RAM.

Type: EUG

Type: D4000,5000 and press RETURN

The "D" stands for display. You are displaying the contents of memory locations \$4000 through \$5000.

Based on what you saw, what is currently stored in locations \$4000-\$5000? -----

What do the zeros represent?

-
2. Now turn to the memory map in Diagram 6 and locate \$4000-\$5000. How is that area of memory labeled on the memory map?

-
3. The areas of memory that are not free to the programmer are called "ROM" (read only memory) and are reserved for the computer. To see the contents of some ROM,

Type: DF000,FFFF

All those numbers flying by are the machine code of the operating system. The operating system contains the machine language programs necessary for you need in order to use the various input devices available with the computer. For example, a program in the operating system interprets a key press on the keyboard.

4. Now load the assembly language version of the message program into memory. First type "x" to get back into the editor and then enter the file.

Type: X

Type: Enter #D:TEXT

5. You should see the word "EDIT" in the upper left hand corner of your screen.

Type: LIST 0,200

6. You should see the assembly language routine we POKEd into memory in the message program. Note that it begins with an asterisk followed by \$600. In assembly language you have to give the hexadecimal address of where you want your program stored in memory. The program was stored on page 6 -- page 6 is free RAM.

7. Type ASM and press RETURN.

The assembly language version is being "assembled" to machine code.

8. Type BUG and press RETURN.

We are going into the "debugger," which will enable us to peer into memory.

9. Type D600,615 and press RETURN.

This will display the contents of memory from \$600 through \$615.

It should look like this:

```
0600  A9 05 85 CD A0 00 B9 11
0608  06 91 58 C8 C4 CD D0 F6
0610  60 28 25 2C 2C 2F
```

The numbers on the left are the hexadecimal addresses of the memory locations. The memory locations are listed in groups of eight. So on the first line the two digits following 600 (A9) are the contents of memory location 600. The next two digits are the byte in \$601.

\$600 = A9
\$601 = 05
\$602 = 85

The second row lists the contents of eight consecutive locations starting at \$608. OOPS! What happened between the second and the third lines where the starting addresses go from 608 to 610? Don't forget, the memory addresses are in hexadecimal. Line 608 lists the contents of memory locations 608, 609, 60A, 60B, 60C, 60D, 60E, and 60F.

10. Store each pair of hexadecimal values, as you see them on the screen, into each memory location of the memory map below. This is how your programs are stored in memory.

\$600		-----	

\$605		-----	

\$60A		-----	

\$60F		-----	

Summary and Challenges

A Memory Address = Two Bytes = \$9C40

\$9C = The high order byte.

\$40 = The low order byte.

The high order byte of an address indicates which page of memory the location is on.

The low order byte of an address indicates where the byte is located in the page of memory.

One page of memory = 256 locations.

One K of memory = 1024 locations.

Four pages = 1K

Take a look at some of the specific memory locations you have POKEd values into in the past. To see the contents of a memory location, you just type "D" and the hexadecimal address in memory.

To change or POKE a new value into a memory location,

Type: C address < new contents

For example, try, C2F3<4

This changes the contents of memory location \$2F3 (530 in decimal) to 4.

In order to return the letters to an upright position,

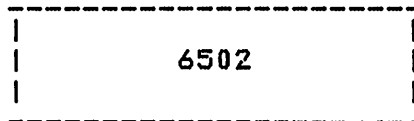
Type: C2F3<2

Take this opportunity to have a look through the Master Memory Map by Santa Cruz Educational Software. Experiment with changing the contents of some locations.

The Central Processing Unit

This section introduces to the 6502 microprocessor. The major components of the 6502 are discussed, and you will have an opportunity to execute your first assembly language program.

At the heart of your Atari computer and all its operations is the 6502 microprocessor. If you could look inside your computer, you would find that the CPU itself is really quite small. The 6502 microprocessor consists of a tiny silicon chip housed in a black plastic box approximately two inches by a half an inch.

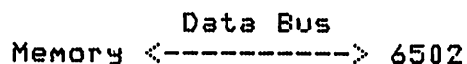


Approximate size of the 6502
Microprocessor in the Atari

The 6502 serves as the master controller or "brain" of the computer. It executes the instructions in your program. However, the 6502 can hold only one instruction at a time -- the instruction it is currently working on. Meanwhile, the rest of the instructions in your program, are safely stored in memory. In order to execute your program the 6502 "fetches" the instructions from memory one at a time.

Although the 6502 works very rapidly, it only can execute one instruction at a time. For example, adding two numbers together is one operation, but it demands all of the CPU's attention until the computation is complete. Memory enables the microprocessor to have easy access to your program so that successive instructions can be completed very rapidly. As calculations are made by the microprocessor, the results are saved in memory. Thus, together the 6502 microprocessor and memory perform the main functions of the computer.

The link between memory and the microprocessor is a complicated set of wires called the "Data Bus."



The 6502 is made up of six major components, which are called "internal registers." A register is a storage location. The registers in the CPU serve different functions.

Diagram 7

6502 Model

----- -----	Accumulator (holds 1 byte)
----- -----	X Register (holds 1 byte)
----- -----	Y Register (holds 1 byte)
----- -----	Program Counter (holds 2 bytes)
\$01 -----	Stack Pointer (holds 1 byte)
----- -----	Processor Status Register (1 byte)

The Accumulator:

In order to transfer data back and forth between memory and the CPU, the value must be passed through one of three registers: the Accumulator, the X register, or the Y register. The accumulator is the most commonly used register for data transfer. An instruction in assembly language, "LDA" or LoaD the Accumulator, instructs the CPU to load the accumulator with the contents of a specified memory location.

The accumulator holds one byte, as does one memory location. Because they are equal in size, data transfer from memory to the accumulator and vice versa is simple.

The X Register and the Y Register:

The X and Y registers each hold one byte and they also can be used to transfer data between memory and the CPU. However, the X and Y registers are more commonly referred to as "index registers," because programmers often use them as counters or "indexes" to a loop. In assembly language you

can repeat instructions in a "loop" as you would in a BASIC program with a "FOR . . . NEXT" loop. To use the X register or the Y register as a counter, the program adds or subtracts one from the number in the index register each time a set of instructions is repeated. The number in the index register is the number of times the loop has been executed.

Turn to Machine Architecture Worksheet #7 for a better look at how the internal registers are used. You will need to use a pencil to complete this worksheet.

Machine Architecture Worksheet #7

Let's look at an example of how the accumulator and the X register are used to execute a simple assembly language program. We don't expect you to understand what each assembly language instruction means in this exercise. Instead, we hope that you will learn more about the machine processing cycle by completing this worksheet. The program multiplies 4 times 5. The microprocessor multiplies by doing a series of additions. For example, to multiply 4*5, the computer adds 5, four times. In assembly language there are no instructions to multiply numbers. The 6502 only knows how to add or subtract.

$$4 \times 5 = 5 + 5 + 5 + 5$$

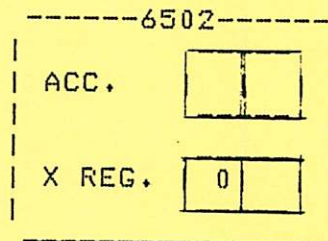
The assembly language routine is listed below.

```
LDX #4      ;LOAD THE X REGISTER WITH COUNT OF 4
LDA #0      ;LOAD THE ACCUMULATOR WITH ZERO
CLC         ;CLEAR THE CARRY
ADD #5      ;ADD 5 TO VALUE IN ACC.
DEX         ;SUBTRACT ONE FROM COUNTER IN X REG.
BNE ADD     ;IF COUNTER ISN'T ZERO ADD AGAIN
STA $060E   ;STORE ACC. VALUE IN MEMORY
RTS         ;RETURN
```

Imagine that we just loaded this program into memory. And you, alias Mike Roprocessor, are about to execute the program. You will step through the program and execute one instruction at a time.

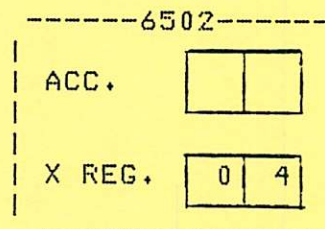
1. LDX #4: Load the X Register with 4, which is stored after the LDX instruction in memory. This sets the counter for the number of additions that will be computed. Load the X register below with 4. Don't worry about the fact that some memory locations are empty.

-----MEMORY-----	
Address	Value
\$600	(LDX)
\$601	4
\$602	(LDA)
\$603	0
\$604	(CLC)
\$605	(ADC)
\$606	5
\$607	(DEX)
\$608	(BNE)
\$609	
\$60A	(STA)
\$60B	
\$60C	
\$60D	(RTS)
\$60E	



2. LDA #0: Load the accumulator with the 0, which is stored in memory after the LDA instruction. This insures that the accumulator is cleared to zero before we begin adding.

-----MEMORY-----	
Address	Value
\$600	(LDX)
\$601	4
\$602	(LDA)
\$603	0
\$604	(CLC)
\$605	(ADC)
\$606	5
\$607	(DEX)
\$608	(BNE)
\$609	
\$60A	(STA)
\$60B	
\$60C	
\$60D	(RTS)
\$60E	



3. We will ignore the CLC instruction to the CPU for now.

4. ADC #5: Now add 5 to the number in the accumulator. This calculation is performed by the CPU. The answer is put in the accumulator. If this is the first time you have executed this instruction, look back to the accumulator in number 2 above to see what is currently in the accumulator. Add 5 and store the new value in box #1 below. Otherwise, fill in the boxes below in the order in which you execute this instruction. If this is the second time you have executed this instruction, add 5 to the value in the accumulator in box #1 and store the new value in the accumulator in box #2. Remember to use hexadecimal numbers.

-----MEMORY-----	
Address	Value
\$600	(LDX)
\$601	4
\$602	(LDA)
\$603	0
\$604	(CLC)
\$605	(ADC)
\$606	5
\$607	(DEX)
\$608	(BNE)
\$609	
\$60A	(STA)
\$60B	
\$60C	
\$60D	(RTS)
\$60E	

#1.	-----6502-----	#3.	-----6502-----
	ACC. <table border="1" style="display: inline-table; width: 60px; height: 20px;"></table>		ACC. <table border="1" style="display: inline-table; width: 60px; height: 20px;"></table>
	X REG. <table border="1" style="display: inline-table; width: 60px; height: 20px; text-align: center;">04</table>		X REG. <table border="1" style="display: inline-table; width: 60px; height: 20px; text-align: center;">02</table>
-----		-----	
#2.	-----6502-----	#4.	-----6502-----
	ACC. <table border="1" style="display: inline-table; width: 60px; height: 20px;"></table>		ACC. <table border="1" style="display: inline-table; width: 60px; height: 20px;"></table>
	X REG. <table border="1" style="display: inline-table; width: 60px; height: 20px; text-align: center;">03</table>		X REG. <table border="1" style="display: inline-table; width: 60px; height: 20px; text-align: center;">01</table>
-----		-----	

5. DEX: An addition has been completed, so the X register is decremented by one. Subtract one from the X register and store the new counter back in the X register. To update the X register, fill in the boxes below in the order in which you execute this instruction. If this is the first time you have executed this instruction, look back at the value in the X register in #4 and subtract one. Put the new value for X in the X register in box #1 below. If this is your second time at this instruction, update the X register in box #2.

-----MEMORY-----

Address	Value
\$600	(LDX)
\$601	4
\$602	(LDA)
\$603	0
\$604	(CLC)
\$605	(ADC)
\$606	5
\$607	(DEX)
\$608	(BNE)
\$609	
\$60A	(STA)
\$60B	
\$60C	
\$60D	(RTS)
\$60E	

#1. -----6502-----

ACC.	0	5
X REG.		

#3. -----6502-----

ACC.	0	F
X REG.		

#2. -----6502-----

ACC.	0	A
X REG.		

#4. -----6502-----

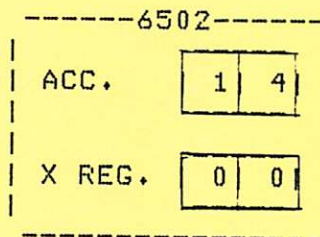
ACC.	1	4
X REG.	1	

6. BNE ADD: Is the number in the X register equal to zero? If not, branch back to #4 and repeat instructions #4 and #5. If the counter in the X register is equal to zero, continue with #7 below.

7. STA \$060E: In order to save your answer, store the accumulator in memory location \$60E.

-----MEMORY-----

Address	Value
\$600	(LDX)
\$601	4
\$602	(LDA)
\$603	0
\$604	(CLC)
\$605	(ADC)
\$606	5
\$607	(DEX)
\$608	(ENE)
\$609	
\$60A	(STA)
\$60B	
\$60C	
\$60D	(RTS)
\$60E	<input type="text"/>



8. RTS: The last instruction in the program, RTS, returns the CPU's attention to the program that called this subroutine.

Congratulations, Mike Roprocessor! You have just executed an assembly language program.

The Program Counter:

The program counter is a 16 bit (2 byte) register in the 6502. The program counter must be two bytes, because it holds an address and all addresses are two bytes. As your program is running, the program counter is continually updated to the address of the next instruction the CPU will execute. The program counter keeps a watchful eye on your program!

The Stack Pointer:

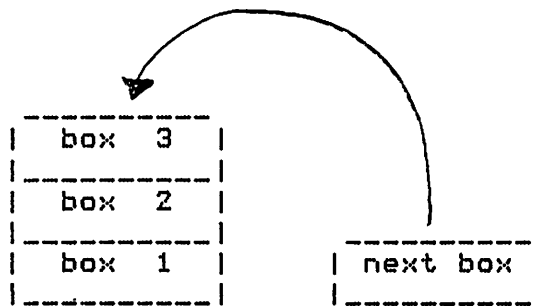
The stack pointer holds the address of the next available location in an area of memory called the "stack." The stack pointer is called a "pointer," because it holds an address and thus is said to be pointing to a location in memory.

The stack is a set of 256 memory locations set aside for temporary data storage. The stack resides in memory locations \$0100 - \$01FF, which is also referred to as "page one" of memory. The high order byte (2 digits on the left) of the addresses on page one of memory are all \$01, (\$0100-\$01FF). Consequently, only one byte is required for the stack pointer.

\$01	00000000	= Stack Pointer
\ /	\ /	
High Byte	Low Byte	
Is constant	A number between 0 and 255 which	
	indicates the location on page one.	

Data is stored on the stack in a very systematic way. A "Last In First Out" (LIFO) filing system is used, which means that the most recent byte of data stored on the stack is always the first byte you get off the stack.

One way to better understand the stack is to think of it as a tower of heavy boxes. As each new box is added to the tower, it is put on top of the stack of boxes.



To get to box #2 in the stack, first the top box and then box #3 must be taken down. Remember, these are heavy boxes, so you can lift only one at a time. No matter which box you want boxes must be removed continually from the top of the pile until the desired box is reached.

Because of the restrictions of the "Last In First Out" filing system of the stack, programmers should carefully plan the order in which they place data on the stack for later retrieval.

Now let's look at an example of the steps the computer takes to complete a program that includes using the stack. Turn to Machine Architecture Worksheet #8.

Machine Architecture Worksheet #8

This time you will execute a program which solves the equation $2 \times (3 \times 13)$. Use a pencil so you can change the values in the registers and memory as you execute the program. The assembly language routine is listed below. Once again, you are not expected to completely understand the assembly language program. The instructions are all explained in the Assembly Language Module. Our intent here is to familiarize you with the microprocessor, and how it executes assembly language programs.

```
      *= $0600      ;STORE PROGRAM STARTING AT $600 IN MEMORY
      SUM = $061C    ;LOCATION FOR SUM
      TOTAL= $061D   ;SAVE MEMORY FOR TOTAL
      CLC           ;CLEAR THE CARRY BIT
      LDY #2        ;LOAD Y REG. TO MULTIPLY BY 2
TWICE LDX #3        ;LOAD X WITH 3 TO COUNT ADDITIONS OF 13
      LDA #0        ;LOAD ACC. WITH ZERO TO START
      ADD ADC #13    ;ADD 13 TO ACCUMULATOR
      DEX           ;DECREMENT X REGISTER BY 1
      BNE ADD       ;IS X REG. = 0? NO, ADD AGAIN
      PHA          ;PUT VALUE IN ACCUMULATOR ON STACK
      DEY          ;DECREMENT Y REGISTER BY 1
      BNE TWICE     ;IS Y REG. = 0? NO, MULTIPLY 3*13 AGAIN
      PLA          ;PUT LAST VALUE PUT ON STACK IN ACC
      STA SUM       ;STORE THE CONTENTS OF ACC IN MEMORY
      PLA          ;PUT FIRST NUMBER PUT ON STACK IN ACC
      ADC SUM       ;ADD SUM TO NUMBER IN ACC
      STA TOTAL     ;STORE TOTAL IN MEMORY
      RTS          ;RETURN FROM SUBROUTINE
```

1. *= \$600: This instruction indicates that the program will be loaded into memory starting at \$600. The next two lines, which contain the variables SUM and TOTAL followed by an equals sign, are instructions which reserve memory locations for each of those variables.

2. CLC: Once again we will overlook the "Clear the Carry bit" command to the CPU.

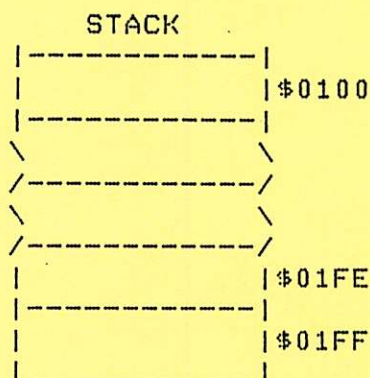
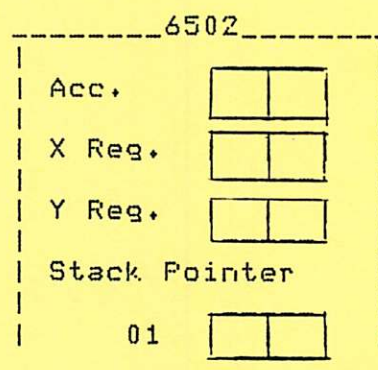
3. LDY #2: Load the Y register with the number 2, which is stored in memory following the LDY instruction. The Y register will serve as a counter for multiplying (3×13) by 2. Put a 2 in the Y register on the following page.

4. LDX #3: Now load the X register with a 3. We are multiplying by 3, by adding 13 three times.

5. LDA #0: Load the accumulator with zero to start.

-----MEMORY-----

Address	Value
\$600	(CLC)
\$601	(LDY)
\$602	2
\$603	(LDX)
\$604	3
\$605	(LDA)
\$606	0
\$607	(ADC)
\$608	13
\$609	(DEX)
\$60A	(BNE)
\$60B	
\$60C	(PHA)
\$60D	(DEY)
\$60E	(BNE)
\$60F	
\$610	(PLA)
\$611	(STA)
\$612	
\$613	
\$614	(PLA)
\$615	(ADC)
\$616	
\$617	
\$618	(STA)
\$619	
\$61A	
\$61B	(RTS)
\$61C	
\$61D	




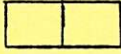
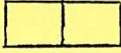
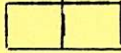
6. ADC #13: Now add 13 to the contents of the accumulator. Simply erase the number already in the accumulator, and replace it with the updated value. As you continue to execute the program, store the values in the registers and memory as hexadecimal numbers.

7. DEX: Decrement the X register by subtracting one from the contents of the X register and storing the new count back in the X register. The number in the X register indicates how many times 13 has been added to the value in the accumulator.

8. BNE ADD: Is the number in the X register zero? If not, branch to #6 and complete #6 and #7 again. By redoing #6 and #7, you are adding on another 13 to the accumulator in order to compute the (3 * 13) part of the equation. When the X register is zero, you are ready to go on to #9. Continue to execute the program, using the memory and CPU on the following page.

9. PHA: "PHA" stands for Push the Accumulator onto the stack. This instruction tells you to store a copy of the value in the accumulator on the stack. The stack fills from the highest address on the stack down to the lowest, with one exception. The first value put on the stack goes into the stack memory location \$0100. The second value put on the stack goes in \$01FF. From then on, values are stored on the stack from the top down (\$01FF down to \$0101). Push the value in the accumulator onto the stack. Remember to use hexadecimal numbers.

The stack pointer holds the address of the next available location on the stack. When the stack is empty, the stack pointer is \$00. The stack pointer must be updated as well. Put \$FF (or \$FE if this is the second time you have executed this instruction) in the stack pointer.

-----MEMORY-----		
Address	Value	
\$600	(CLC)	<div>6502</div> <div>Acc. </div> <div>X Reg. </div> <div>Y Reg. </div> <div>Stack Pointer</div> <div>01 </div>

10. DEY: To decrement the Y register, subtract one from the contents of the register. The Y register holds the counter for the number of times (3×13) has been calculated.

11. BNE TWICE: BNE stands for "branch not equal to zero." Is the number in the Y register equal to 0? If not, then branch back to #4 and re-execute instructions #4 - #10. If the Y register is zero, then continue with #12.

12. PLA: "PLA" is the opposite of a PHA. Pull off the last value put on the stack and put it in the Accumulator. (Erase the value being pulled from the stack and record the value in the accumulator.) Remember to update the stack pointer. This time add one to the stack pointer, so that it points to the next available location.

13. STA SUM: Store the contents of the accumulator in memory location \$61C, which has been reserved for the variable SUM. This will enable us to save the SUM of $13+13+13$ for later use.

14. PLA: Now pull the first value you stored on the stack off and put it in the accumulator. Erase the value in the stack, record it in the accumulator, and update the stack pointer. Since there is nothing left on the stack, the stack pointer gets reset to \$0100.

15. ADC SUM: Add the contents of the memory location \$61C to the contents of the accumulator. The two products of (3×13) are being added. Again the computer is completing a multiplication problem by adding. Update your accumulator with the result of the addition.

16. STA TOTAL: Store the contents of the accumulator in memory location \$61D, the memory location which has been reserved for the TOTAL. The answer to $2 \times (3 \times 13)$ is now stored in memory and ready for any further use. Depending on what instructions followed this subroutine, the answer could be printed on the screen, added to another number, etc.

17. RTS: Return to BASIC or the assembler from this subroutine.

How do you feel, Mike Roprocessor? You have just completed another assembly language program. Did that seem like a lot of work for solving $2 \times (3 \times 13)$? Amazingly enough, your friend 6502 could execute the same program in a split second!

Follow the next 6 instructions to observe the computer execute the same program you just executed by hand. You will need the Assembler Editor Cartridge and the Advanced Topics Diskette to do this. Put the Assembler cartridge in the slot on the left where your BASIC cartridge ordinarily goes, and boot your diskette.

1. You should have the EDIT prompt in the upper left hand corner of your screen. Load the "EQUATION" file on your Advanced Topics diskette.

Type: Enter #D:EQUATION

2. Type LIST. You should see the program you just executed by hand listed on the screen.

3. Your program needs to be converted to machine code by the assembler and stored in memory.

Type: ASM

4. In order to look at the contents of specific memory locations and the registers, you need to get into the debugger.

Type: BUG

5. The debugger also enables us to "step" through the program and observe the computer executing one instruction at a time. The first instruction of the program is at \$600.

Type: S600

The "S" stands for step. At the bottom of the screen you should see the machine code contained in memory location \$600, the corresponding assembly language instruction, and the contents of the internal registers as shown below.

```
600      18      CLC
      A=00  X=00  Y=00  P=30  S=00
```

The 600 is the address of the memory location. The 18 is the machine code for the assembly language instruction CLC. The next line lists the contents of the internal registers. The A, X, and Y are self-explanatory. The P stands for the Processor Status register, which we will cover in the next section, and the S represents the stack pointer.

6. Now Type: S and press return.

The LDY #2 instruction was just executed and the registers were updated. Step through the program by typing "S" and RETURN after each instruction has been executed. Compare your total with the number in the accumulator when you reach the RTS instruction. The answer or TOTAL was stored in memory location \$61D. To see what is stored in \$61D,

Type: D61D and press RETURN

The "D" stands for display, followed by the memory location you wish to see. Is the computer's TOTAL the same as yours?

7. To see how fast the computer executes the EQUATION program,

Type: G600

The "G" stands for "GO" or execute the program which is stored in memory starting at \$600. The RTS at the end of the EQUATION program returns the computer to the assembler, and consequently there is a break at \$0001.

Processor Status Register:

The status register is also a one byte register. However, instead of the byte holding a number such as an address, each of the eight bits of the byte means something different. For example, one bit indicates if there is a negative number in one of the internal registers. Another bit indicates if there is a zero in one of the registers. The processor status register information is based on the results of the 6502's most recent computation. Diagram #8 shows the significance of each bit. Don't worry if you don't fully understand what the status bits are at this point.

Diagram #8

7	6	5	4	3	2	1	0
0	0		0	0	0	0	0
N	V		B	D	I	Z	C

N = Negative Result. Indicates whether the result of an arithmetic operation was a negative number.

V = Overflow. Indicates whether the result of a mathematical calculation was larger than 255, the maximum number which can be stored in one byte.

An unused bit.

B = Break Command. Indicates whether there has been a break in the 6502's processing.

D = Decimal Mode. Controls whether the math operations will be computed in binary or decimal mode.

I = Interrupt Disable. Controls the interruptions to the 6502's processing.

Z = Zero Flag. Indicates whether the result of the most recent calculation was a zero.

C = Carry Flag. If the answer to an addition problem is greater than 255, which is the largest quantity an 8 bit byte can hold, the carry bit is set.

Each of the eight bits is comparable to an on/off switch. If the bit is a "one," it is said to be "set" or on. A "zero" indicates that the condition does not exist (the switch is off) and the bit is "clear." The bits of the status register are referred to as "flags."

Suppose you just calculated 5-7. The result is negative, and consequently the sign flag (bit 7) of the status register will be set. (Disregard the other flags set by this computation. They will be discussed later.)

Status Register

7	6	5	4	3	2	1	0
1	0	1	1	0	0	0	0
N	V		B	D	I	Z	C

↑

Suppose the most recent calculation was 7-7. As a result, the sign flag would be clear, but the zero flag (bit 1) would be set to 1.

Status Register

7	6	5	4	3	2	1	0
0	0	1	1	0	0	1	0
N	V		B	D	I	Z	C

↑

Turn to Machine Architecture Worksheet #9, to have a look at the contents of some of the registers and how they are displayed.

Machine Architecture Worksheet #9

You will need an Assembler Editor Cartridge and your Advanced Topics Diskette to complete this worksheet.

1. Boot up your system. The EDIT prompt should be in the upper left hand corner of the screen.

Type ENTER #D:EQUATION and press RETURN

2. The EDIT prompt is still in the upper left hand corner. Now,

Type: ASM

You will see the assembly language version of the program going by on the screen as it is being converted to machine code.

3. Now Type BUG. This puts you in the debugger, where you can look at what is stored in the registers. You should see the DEBUG prompt on the screen.

4. To run the program EQUATION,

Type: G600

This stands for GO \$600, which runs the assembly routine from its starting address at \$600. At the bottom of the screen, the contents of the registers should be displayed. The program stops because of the RTS at the end of the program. If the program had been called from a BASIC program, it would have gone back to the BASIC program. Since you ran the program from the assembler, the RTS returns you to the first byte of memory, \$0001, where the program breaks and the registers are displayed.

5. Record the values of the different registers below.

Accumulator	X Reg.	Y Reg.	Processor Status Reg.	Stack
-------------	--------	--------	-----------------------	-------

A =	X =	Y =	P =	S =
-----	-----	-----	-----	-----

Note that the values are listed in hexadecimal.

6. Convert the value in the Processor Status Register to an eight bit binary number.

P = \$_____ = _____ in base 2.

7. According to the binary byte you got, which of the status register flags are clear and which are set?

N V B D I Z C

Set Clear (Check One)

N = Negative flag

V = Overflow flag

B = Break command

D = Decimal mode

I = Interrupt disable

C = Carry flag

The unused flag and the break flag are set whenever a program is run.

Summary and Challenges

The 6502 central processing unit contains six major components:

1. Accumulator
2. X Register
3. Y Register
4. Program Counter
5. Stack Pointer
6. Processor Status Register

Executing a program involves a repeated transfer of data back and forth between memory and the 6502 microprocessor.

To learn more about machine architecture and assembly language, you may wish to read The ATARI Assembler by Don and Kurt Inman, which is available in the camp library. Plan to proceed with the Assembly Language module to learn more about the assembly language instruction set and how to program in assembly language.

Challenges

1. Write and hand-process an assembly language program to count to 100 by 10.
2. Write and execute a program that solves the equation $(5 * 8) + 3$.

Additional Chips

In addition to the 6502 microprocessor, the ATARI has the Antic, GTIA, and Pokey chips, which enhance the graphics, and sound of the computer. Each chip is explained briefly below.

Antic:

The Antic chip is a microprocessor devoted entirely to handling television display. The Antic is programmable, just as the 6502 is. The programs written for the Antic chip are called display lists. To find out more about the Antic chip and display lists, see the Display List Module.



































GTIA:

The GTIA chip, a fairly recent addition to the Atari computer, replaced the old CTIA chip. The GTIA chip increases the range of colors available to the programmer and offers three additional graphics modes (9,10,11). Antic controls most of the GTIA's operations.

POKEY:

POKEY performs a number of functions involving input and output. It handles the transfer of information between memory and the CPU, and along the Data Bus, as well as sound, reading the keyboard, and random number generation.

Table 9.6—INTERNAL CHARACTER SET

Column 1			Column 2			Column 3			Column 4						
#	CHR	#	CHR	#	CHR	#	CHR	#	CHR	#	CHR	#	CHR		
0	Space	16	0	32	@	48	P	64		80		96		112	p
1	!	17	1	33	A	49	Q	65		81		97	a	113	q
2	"	18	2	34	B	50	R	66		82		98	b	114	r
3	#	19	3	35	C	51	S	67		83		99	c	115	s
4	\$	20	4	36	D	52	T	68		84		100	d	116	t
5	%	21	5	37	E	53	U	69		85		101	e	117	u
6	&	22	6	38	F	54	V	70		86		102	f	118	v
7	'	23	7	39	G	55	W	71		87		103	g	119	w
8	(24	8	40	H	56	X	72		88		104	h	120	x
9)	25	9	41	I	57	Y	73		89		105	i	121	y
10	•	26	:	42	J	58	Z	74		90		106	j	122	z
11	+	27	;	43	K	59	[75		91 ⁰		107	k	123	
12	,	28	<	44	L	60	\	76		92		108	l	124	
13	—	29	=	45	M	61]	77		93		109	m	125 ⁰	
14	▸	30	>	46	N	62	^	78		94		110	n	126 ⁰	
15	/	31	?	47	O	63	—	79		95		111	o	127 ⁰	

1. In mode 0 these characters must be preceded with an escape, CHR\$(27), to be printed

Hexadecimal to Decimal Conversion

To convert a hexadecimal number using the chart below, use the vertical digits on the border of the chart to represent the high order nybble of your hexadecimal byte. The horizontal digits on the border of the chart represent the low order nybble of the hexadecimal byte. So if you want to convert \$10 to a decimal number, first look for 1 on the vertical number line, and then look for 0 on the horizontal numbers. Follow the two digits towards the center of the matrix and you find that \$10=16.

Hex and Decimal Conversion

LSD—	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	
0	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	0
1	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	1
2	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	2
3	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	3
4	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79	4
5	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95	5
6	96	97	98	99	100	101	102	103	104	105	106	107	108	109	110	111	6
7	112	113	114	115	116	117	118	119	120	121	122	123	124	125	126	127	7
8	128	129	130	131	132	133	134	135	136	137	138	139	140	141	142	143	8
9	144	145	146	147	148	149	150	151	152	153	154	155	156	157	158	159	9
A	160	161	162	163	164	165	166	167	168	169	170	171	172	173	174	175	A
B	176	177	178	179	180	181	182	183	184	185	186	187	188	189	190	191	B
C	192	193	194	195	196	197	198	199	200	201	202	203	204	205	206	207	C
D	208	209	210	211	212	213	214	215	216	217	218	219	220	221	222	223	D
E	224	225	226	227	228	229	230	231	232	233	234	235	236	237	238	239	E
F	240	241	242	243	244	245	246	247	248	249	250	251	252	253	254	255	F
	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	

```

10 ;PRINT A MESSAGE TO THE SCREEN BY
20 ;PLACING THE INTERNAL CHARACTER
30 ;SET VALUE DIRECTLY IN SCREEN
40 ;MEMORY      FILE: TEXT
50 ;*****
60 ;
0000      0100      *=      $0600
0600 A905      0110      LDA      #$5      ;MESSAGE LENGTH
0602 85CD      0120      STA      $CD
          0130      ;
          0140      ;$CD IS A FREE BYTE ON THE ZERO PAGE.
          0150      ;THE MESSAGE LENGTH IS BEING STORED THERE
          0160      ;
0604 A000      0170      LDY      #00      ;COUNTS EACH LETTER AS OUTPUT
0606 E91106    0180 LETTER LDA      $611,Y      ;GET THE NEXT LETTER
0609 9158      0190      STA      ($58),Y      ;PUT LETTER ON NEXT SCREEN LOCATION
060B C8        0200      INY      ;INCREMENT LETTER COUNTER
060C C4CD      0210      CPY      $CD      ;THE END OF THE MESSAGE?
060E D0F6      0220      BNE      LETTER      ;NO, GET ANOTHER LETTER
0610 60        0230      RTS
0611 28        0240      .BYTE 40,37,44,44,47
0612 25
0613 2C
0614 2C
0615 2F

```

```

100 REM *   A PROGRAM TO FILL THE SCREEN WITH ONE INPUT CHARACTER
110 REM *
120 REM *****
130 REM *
140 DIM A$(1):REM DIMENSION INPUT STRING
150 PRINT "PRESS ANY KEY";
160 INPUT A$:REM LETTER TO FILL SCREEN
170 FOR I=1 TO 874
180 PRINT A$;:REM OUTPUT CHARACTER
190 NEXT I
200 GOTO 150

```

```

10 REM *                               FILLSCREEN
20 REM *
30 REM *  A PROGRAM WHICH FILLS THE SCREEN WITH ONE LETTER
40 REM *  ACCORDING TO THE MOST RECENT KEYPRESS.  AN ASSEMBLY
50 REM *  LANGUAGE ROUTINE IS POKED INTO MEMORY STARTING AT
60 REM *  1536 ($600) USING THE DECIMAL VALUES FOR THE MACHINE
70 REM *  CODE LISTED IN DATA LINES 220-250.  THE PURPOSE
80 REM *  OF THIS PROGRAM IS TO DEMONSTRATE THE SPEED OF AN
90 REM *  ASSEMBLY LANGUAGE ROUTINE.
95 REM * *****
100 REM *
110 REM *  LINES 140-180 READ THE ASSEMBLY ROUTINE
120 REM *  DATA AND POKE IT INTO MEMORY
130 REM *
140 PROGRAMLEN=74:REM ASSEMBLY ROUTINE IS 75 BYTES LONG (0-74)
150 FOR CODE=0 TO PROGRAMLEN
160 READ INSTRUCTION
170 POKE 1536+CODE,INSTRUCTION
180 NEXT CODE
190 REM *
200 REM *  ASSEMBLY ROUTINE DATA
210 REM *
220 DATA 104,104,104,141,77,6,201,0,240,23,201,32,48,4,201,95,48,9,24,105
230 DATA 64,141,77,6,76,33,6,56,233,32,141,77,6,165,88,133,203,165,89,133
240 DATA 204,169,3,141,76,6,169,152,141,75,6,173,77,6,160,0,145,203,230
250 DATA 203,208,2,230,204,206,75,6,208,243,206,76,6,16,238,96
260 PRINT "PRESS ANY KEY";
270 OPEN #2,4,0,"K:"
280 GET #2,CHARACTER
290 REM *  CALL EXECUTES THE ASSEMBLY ROUTINE IN MEMORY
300 CALL=USR(1536,CHARACTER)
310 GOTO 280

```

```

0100 ; FILLS THE GRAPHICS 0 SCREEN
0110 ; WITH A CHARACTER PASSED FROM
0120 ; A BASIC PROGRAM.
0130 ; FILE : FILL
0140 ;*****
0150 ;
0000 0160 *= $CB
00CB 0170 SCREEN = *
00CB 0180 *= $0600
0600 A941 0190 LDA #65 ;GET CHARACTER VALUE,A
0602 8D4C06 0200 STA CHR ;SAVE CHARACTER HERE
0210 ;
0220 ;CONVERT THE ATASCII VALUE TO THE INTERNAL CHARACTER SET VALUE
0230 ;
0240 ;
0605 C900 0250 CMP #0 ;SPACE
0607 F017 0260 BEQ BEGIN ;DISPLAY CHARACTER
0609 C920 0270 CMP #32 ;IS ATASCII 32 OR LESS?
060B 3004 0280 BMI ADD ;YES, ADD 32 FOR INTERNAL CHARACTER SET
060D C95F 0290 CMP #95 ;IS ATASCII 95 OR LESS?
060F 3009 0300 BMI SUB ;THEN SUBTRACT 32 FOR INTERNAL CHAR VALUE
0611 18 0310 ADD CLC ;CLEAR THE CARRY
0612 6940 0320 ADC #64 ;ADD 64
0614 8D4C06 0330 STA CHR ;STORE INTERNAL CHARACTER VALUE IN CHAR
0617 4C2006 0340 JMP BEGIN ;DISPLAY
061A 38 0350 SUB SEC
061B E920 0360 SBC #32 ;SUBTRACT 32, ATASCII TO INTERNAL CHARS
061D 8D4C06 0370 STA CHR
0380 ;
0390 ; SET UP SCREEN RAM AND OUTPUT COUNTERS
0400 ;
0620 A558 0410 BEGIN LDA $58 ;START ADDRESS SCREEN RAM
0622 85CB 0420 STA SCREEN ;FREE LOCATION ON ZERO PAGE
0624 A559 0430 LDA $59 ;HIGH BYTE SCREEN RAM
0626 85CC 0440 STA SCREEN+1 ;FREE LOCATION ON ZERO PAGE
0628 A903 0450 LDA #03 ;HIGH COUNT VALUE
062A 8D4B06 0460 STA CNT2
062D A998 0470 LDA #152 ;LOW COUNT VALUE
062F 8D4A06 0480 STA COUNT
0490 ;
0500 ; LOOP TO FILL THE SCREEN
0510 ;
0632 AD4C06 0520 LDA CHR
0635 A000 0530 LDY #00 ;INDEX
0637 91CB 0540 FILL STA (SCREEN),Y ;DISPLAY
0639 E6CB 0550 INC SCREEN ;INCREMENT ADDRESS LOW BYTE
063B D002 0560 BNE SKIP1 ;IF NOT 0,BRANCH TO SKIP1
063D E6CC 0570 INC SCREEN+1 ;ADD TO HIGH BYTE
063F CE4A06 0580 SKIP1 DEC COUNT ;COUNT BYTES DONE
0642 D0F3 0590 BNE FILL ;IF NOT ZERO, FILL
0644 CE4B06 0600 DEC CNT2 ;16 BIT ARITH

```


0647	10EE	0610	BPL	FILL	;UNTIL DONE
0649	60	0620	RTS		;DONE - RETURN TO BASIC
		0630	;		
		0640	; DATA STORAGE AREA		
		0650	;		
064A		0660	COUNT	*=	*+1 ;HIGH COUNT VALUE
064B		0670	CNT2	*=	*+1 ;
064C		0680	CHR	*=	*+1 ;CHARACTER VALUE